摘要:
大多数的编码工程项目有前端和后端两个部分。前端部分有着和用户进行交互的任务，对于一个项目，可能前端部分不必过于精美，但是必须要有必要的基础功能。然而编写前端代码需要对 HTML、CSS、JavaScript 等基础知识有一定的了解，进阶的框架更是不计其数。然而对于一个没有接触过前端的开发人员，学习多种新的语言是需要学习成本的，对于没有基础的人，想要快速上手有时候也会显得困难。也正如此，以往的开发方式会把前端和后端明确的分开，开发人员可以专注于自己的任务。然而，对于 C 语言程序员想要快速使用 JavaScript 语言实现简单的网页开发这种情况，本文针对上述特殊情况，提出了一个构想——设计一个类似 C 语言的编程语言，这种语言拥有 C 语言和 JavaScript 的特性，使其能够在浏览器上运行。本文通过设计特定的文法，对源程序进行词法分析，自顶向下的语法分析，语义分析的步骤使得浏览器可以识别源程序， 实现了一个类 C 语言的 JavaScript 解释器。本文从一定程度上减少了开发人员了解学习 JavaScript 的学习成本，对于只需要简单的网页呈现需求的开发人员具有极大的使用价值，同时也为 C 语言跨平台方案提供了一个可行的方向。

第一章 绪论

1.1 研究背景和意义

随着技术的不断发展，如今，越来越多的编程语言涌现到开发人员的视野当中。对于不同的编程语言而言，肯定有其独特的优势，才会有人去使用它进行编程。然而对于种类繁多的语言，每一个语言都去学习并掌握需要花费大量的时间成本，对于学习 C 语言的人群来说，JavaScript 等用于前端的语言未必是他们所擅长的。但是往往很多的 C 语言的项目需要使用前端来进行简单的展示，比如对于学生这些人群来说，大学期间，大多数学习计算机的学生使用的是 C 语言进行编程，学习 Web 等前端的课程不一定是必学的，而每个项目使用控制台进行输入输出难免会使得一些想要把用户的输入和输出页面编写得更加美观的同学变得不满足，而学习前端语言可能会没有足够的时间。所以迫切需要一种能够在浏览器环境运行的类 C 语言，因此基于 JavaScript 的 C 语言解释器是具有极大的研究价值的。

随着电子产品的不断普及，浏览器早已是日常生活中不可或缺的一部分。人们利用浏览器进行信息检索、知识学习、社交购物等，而且手机上都带有浏览器，可见其对人类生活的重要性。因此可以把浏览器也看成一个很重要的平台，而对于一个平台来说，使得一个平台能够更受欢迎的方法是使得更多的东西能够兼容这个平台，而 C 和 C++都作为是排名靠前的编程语言，如果 C 语言能够兼容浏览器，那么 C 语言程序就能在浏览器上开发一些网页，这种新的想法会使得更多的 C 语言开发人员能够对浏览器进行开发，编写出更加多样化的项目，同时也开创了一个新的前端思路，这种创新有很大的研究价值，其为开发人员提供的便利性又使其具有很大的使用价值。

1.2 国内外研究现状

对于基于 JavaScript 的 C 语言解释器，国内外研究得并不是很多，网络上有少量的基于 JavaScript 的 C 语言编译器，但是因为 C 语言的语法之多，网上的项目也仅仅是完成了一小部分的 C 语言库的编写，其功能也是具有很大的局限性的。

1.3 论文的主要工作

本文设计了一种类 C 语言——扩展的 C 语言，该语言具有 C 语言的一些特性，同时引入 let 等 JavaScript 特有的变量类型，结合了 JavaScript 的优点，该语言不进行类型检查，直到运行时才会进行判断，使得该语言更加符合浏览器执行的习惯，而不完全遵循 C 语言的特性。本文以该类 C 语言作为研究对象，针对其设定的文法，使用 JavaScript 对其进行词法分析、语法分析和语义分析。具体的过程是将源程序分析为一组 token 串，然后根据得到的 token 串进行语法分析，最后根据语法树分析程序要执行的语义规则。另外还需要建立符号

表来保存全局变量、临时变量，以及设计语法树和符号表的数据结构。完成上诉工作，使得类 C 语言代码能够在浏览器中正确的运行，展示出程序应表现出的正确结果。

1.4 论文的组织结构

本文内容分为 4 章，每一章的内容安排如下：

第一章“绪论”，介绍本文的研究背景以及意义，对国内外的研究现状进行简单阐述，介绍本文的主要工作。

第二章“预备知识”，对程序的编译过程，解释器和编译器的区别，浏览器主要语言 JavaScript 和浏览器的特殊性进行较为详细的说明。

第三章“项目的设计流程”，介绍完成项目中用到的数据结构，包括语法树、符号表的设计等。同时对文法的设计、词法分析、语法分析、语义分析的细节进行详细的介绍。

第四章“项目的效果展示”，展示项目中完成的主要功能的效果

第五章“总结和展望”，对整个项目进行总结，反思其中的不足，提出能够改善的地方。同时提出对项目提出展望。

第二章 预备知识

2.1 程序的编译过程

一个编译器就是一个程序，编译分为几个特定的步骤：

- 词法分析。词法分析器从源程序中读入字符流，并将它们组织成为有意义的词素的序列。一般来说，词法分析器会产生<token-name,value>形式的词法单元（token）作为输出，传递到下一阶段。

- 语法分析。语法分析从词法分析器的输出中获取 token 串，将 token 的第一个元素作为结点创建树形的中间表示。一种比较常见的形式是语法树。

- 语义分析。语义分析根据语法树和符号表的信息来检查源程序是否和语言定义的语义一致。同时也收集类型信息，将这些信息存放在语法树和符号表中，供后面的过程使用。

- 中间代码生成。在一个源程序翻译成目标代码的过程中，编译可能会生成多种中间表示。语法树就是其中的一种表现形式。中间代码生成主要任务是根据语法树，将源程序表示为一种低级语言或者是类机器语言。但是它需要保证是易于生成的，而且容易翻译成目标机器的语言。

- 代码优化。代码优化意味着对生成的中间代码进行一些必要的优化，使其运行更快，损耗更小。

- 代码生成。代码生成器以已经优化后的中间代码作为输入，将其映射成为目标机器的语言。如果目标语言是机器语言，还需要考虑内存分配，以及每个变量分配的寄存器等因素。将中间代码翻译成为具有相同效果的机器指令进行执行。

2.2 编译器和解释器

编译器和解释器的过程有着很多相似之处。一般来说，编译器和解释器都有词法分析和语法分析的过程。但是在语义分析的过程中，解释器将根据目标语言直接执行语句。而编译器的语义分析过程需要分析其源程序的语义动作，之后将其翻译成中间代码的形式。

编译器一般会将源程序翻译成低级的语言或者是机器语言，解释器不通过翻译的方法生成目标程序，而是根据语法树和符号表可以用目标语言直接执行源程序对应的语句，它的目标语言可以是高级语言也可以是低级语言。

2.3 JavaScript 简单介绍

JavaScript 是一种直译式脚本语言，是一种弱类型、动态类型、基于原型的语言，其解

释器被称为 JavaScript 解释器，是浏览器内部的一部分。

JavaScript 拥有不同的模块化规范，比如 commonJS、AMD、CMD、ES Modules 各个规范的不同是因为开发的环境不尽不同。

第三章 项目的设计流程

3.1 项目设计的目的以及预期效果

本项目设计的目的是使得 C 语言开发人员在不熟悉网页开发的前提下，能够通过使用本项目，在短时间内快速书写简单的网页逻辑和最基础的展示效果。

本项目的预期效果是本文设计了一个基础架构，该架构仅支持最基础的语法，支持函数调用、输入输出、变量定义、简单赋值、计算求值等功能。但是该架构需要是一个可扩展的架构，开发人员可以通过增加文法、添加对应的处理逻辑，可以完善该架构尚未支持的功能。

3.2 项目结构

3.2.1 项目结构概述

本项目由两部分组成，一是前端网页部分，提供一个网页，用户在网页上可以编辑符合规定的代码，点击“提交”按钮即可执行。二是代码解释器部分，该部分由 JavaScript 语言编写，采用的是浏览器端规范。浏览器的主要任务是和用户交互，因此浏览器的模块加载是异步的，也就是尽可能保证浏览器不会在一个任务中阻塞，进而导致浏览器产生“假死”的现象，保证用户的使用体验。

3.2.2 网页前端部分

网页前端部分是一个 HTML 页面，引入代码解释器部分的 JavaScript 文件。主要任务是和用户交互，提供给用户使用本项目的渠道。

3.2.3 代码解释器部分

代码解释器部分是本项目的重点内容，主要的逻辑本质都在这一部分实现。主要包括设计文法、词法分析、语法分析、语义分析。

3.3 项目设计实现

3.3.1 设计文法

本项目设计的文法是上下文无关文法。

什么是上下文无关文法？在应用推导一个产生式时，已经推导出的部分结果就是上下文，而上下文无关的意思是一个非终结符的推导，不依赖于其已产生推导结果。通俗来说，也就是一个文法的左部只能有一个非终结符。比如，

S -> ab

是一个上下文无关文法。而

aSb -> aabb

是一个上下文相关文法。因为这个产生式的左边不仅仅只有一个符号，因此将 S 继续推导的时候，必须确保 S 有正确的“上下文”。

考虑到语法分析的时候采用的是自顶向下的分析方法，因此文法必须要求是非左递归文法。

S ->Sa

S -> b

是一个左递归文法，如果使用自顶向下的分析方法，会陷入死循环中知道程序崩溃。因此需要进行消除左递归。对于上面的左递归文法，可以将其改成如下文法：

S -> bA

A -> aA

A -> null

同样是一个表示已 b 开头后面接 n 个 a 的串的文法，下面的写法是消除左递归的，就可以使用自顶向下的方法进行分析了。

由于本项目的文法数量比较多，不合适全部进行展示，这里仅仅展示一小部分，如下：

AllParams -> null

AllParams -> Param Params

Params -> , Param Params

Params -> null

Param -> Type id

Param -> id

Param -> Type arrayname [ ]

Param -> arrayname [ ]

Type -> int

Type -> let

上述文法是关于参数的文法定义，可以适用于函数定义时的所有参数，也可以适用于函数调用时的所有参数。因此“int a, int b, let c, let array[]”串和“a, 5, array”这样的串都能从上述文法推导得到。项目中还有一些其他的文法，包括变量定义、赋值语句、布尔表达式、函数定义、函数调用、For 循环等，不进行一一说明。

3.3.2 词法分析

词法分析的主要任务是将源程序扫描成字符流进行分析，然后生成 token 串输出。因此我们需要知道词法单元的第一个元素都分为有什么样的类型。

关键字，一般而言进行变量的定义在规则以内都是可以自由选择的，然而每个编程语言都会有保留关键字，比如 C++里面的 int、for、while 和 JavaScript 里面的 function 等等，关键字在编程语言的编译里面有其重要的作用，同时为语法制定了一定规范。进行语法分析的时候需要将关键字提取出来，一个“function”关键字在词法分析后应该得到下面的词法单元：

<function, function>

运算符比如“+”，界限分隔符比如“{”等特殊符号在词法分析后应该得到的是：

<+,+> 和 <{,{>

对于在编程程序中定义的变量，可以采用一个特定的格式：

<id, variable-name>

variable-name 就是定义的变量名，在本文中，对于源程序中出现的数字，同样是使用“id”作为其词法单元的第一个元素。当然如果需要把词法分析做得更加精确，可以输出为如下的词法单元：

<number, 5>

只需要判断输入的字符串是 number 类型即可。在本文中这一部分留到语义分析中才去判断，因此数字其词法单元的第一个元素是“id”。

确定词法单元的正确输出形式之后，就可以对源程序进行读取。本项目中采取数组作为数据结构，这里以“let print = console.log;”作为例子。

1. 读取“l”，数组加入 l
2. 读取“e”，数组填入 e
3. 读取“t”，数组填入 t

4. 读取“ ”（空格），数组内容全部取出，得到“let”，let 是关键字，得到<let，let>，清空数组
5. 读取“p”，数组填入 p
（省略）
6. 读取到第二个空格，取出数组内容，得到“print”，“print”不是关键字，得到<id，print>，清空数组
（省略）
7. 读取到第三个空格，得到<=,=>
.....（省略）
然而词法分析并没有像上述步骤那么简单就能完成，还有一些特殊情况要进行处理：
1. 特殊的运算符，比如“+=”，如果不进行特殊处理，那么读取到“=”的时候，“+”已经作为一个词法单元输出了，因此对于这种情况，需要进行特殊的处理，可以采用向前看一个字符的方式解决，读取到“+”的时候，继续扫描下一个字符“=”，这样就能判断出这是一个两个字符的运算符，避免直接将“+”输出。
2. “.”分隔符，假如源程序中出现“console.log”，“window.alert”等字符流，需要将其整体作为一个变量。这样做的原因是使得文法更加简单。
3. 注释的处理，本项目注释的处理延续 C 语言的习惯，有“//”和“/* ...*/”两种方式使用注释。处理的方式和特殊情况 1 有相似之处，读取到“/”这个字符的时候，需要向前看一个字符。如果下一个字符是‘/’，说明知道换行符之前都是注释，因此无需往数组中填入字符。如果下一个字符是“*”，则证明遇到“*/”之前全都是注释。

3.3.3 语法分析

语法分析其实有很多的方式，但早在 3 本章第 1 节设计文法中就提到的自顶向下的分析方法，这种方法不一定是最优的，由于本项目规模比较小，文法的数量比较少，因此自顶向下的分析速度上不会很慢。

在实现语法分析的过程中，采用的是回溯的方法，使用的数据结构是数组和语法树。将所有的 token 中的第一个元素都按照顺序放到一个数组 A 中，同时新建一个数组 B，该数组初始化的时候填入文法的开始符号，本项目中为“Program”。每一次将 A 和 B 进行比较，找到 A 和 B 两个数组中首个不相同的元素 element，从所有的文法中找到以 element 为左部的文法，将其右部的每个元素填充到元素 element 的位置，因此数组 B 的长度有可能增长也有可能没有发生变化。同时需要将此次使用的文法进行标记，保证回溯的时候不会重复使用同一个文法规则。如果没有左部为 element 的文法，则说明使用的规则不对，需要进行回溯。因此每一次的往下推导的时候需要记住使用的是哪一个文法，需要需要进行回溯的时候需要反向修改数组 B 的内容。如果 A 和 B 能够完全相同，则推导完毕。而如果一直回溯数组 B 到原始的状态，则证明推导失败。

由于每一次的推导，都会将使用到的文法保存在一个数组 C 里面，因此当成功推导完毕的时候，数组 C 里面的每一次推导都是正确的。因此可以根据数组 C 的文法，构建语法树。

语法树的结点的数据结构如下：

- text：结点的文法值
-child_cnt：结点的子结点个数
-child_list[]：结点的子结点的文法值
-tree_list[]：结点的子结点的指针
-val：结点的值

-returnVal：结点的返回值，用于函数返回

读取数组 C 的每一个元素，按照文法的右部元素的数量，新建子节点并加入到对应的数组中去。这样一棵语法树就搭建好了，语法分析的任务就完成了。

3.3.4 语义分析

这里的语义分析不是严格意义上的语义分析。本项目在该阶段对语法分析得到的语法树进行执行，根据结点来执行一个语句应该达到的效果。与编译器不同的是不需要进行中间代码的生成，在该项目中直接使用 JavaScript 来执行语法树的内容。

在执行该语法树的时候，因为代码解释器是基于 JavaScript 语言的，因此无需考虑执行时对象的内存分配等问题。

而对于局部变量和非局部变量的访问，这里使用一个树形的存储方法。设计了一个名叫 Variables 的一个类，这个数据结构成员变量如下：

- type[]：变量的类型数组
- name[]：变量的名字数组
- val[]：变量的值数组
- r1：函数调用时，保存其返回类型
- r2：函数调用时，保存其参数数组
- father：保存其父节点

在程序执行的过程，创建一个保存全局变量的实例，将全局变量保存在该实例中，一般情况下，函数的定义也是属于保存在全局变量中的，因为自身定义的函数可以在任意时候调用。因此需要增加 r1 和 r2 两个值来保存函数的返回类型和函数的参数数组。

假如当执行到一个函数的内容时，这时候需要新建一个类实例，同时将该实例的地址值赋值给创建该实例环境下的实例的 father 变量。这里可以会觉得有点混乱，可以举个实例进行说明，假如有以下代码：

```
int a;
int main () {
    int b,c;
}
```

当执行“int a;”语句时，a 应该保存于刚开始创建的 Variable 实例 v1 中，当执行到 main 函数时，因为正在执行一个函数，函数里面的变量是局部变量，只有在 main 函数中才能访问，因此需要新建另一个 Variable 实例 v2，同时需要另 v2.father = v1，这样做的原因就是在 main 函数中同样是可以访问变量 a 的，因此如果在 v2 中找不到某个变量的值，可以不断地通过 father 找到祖先的 Variable 实例查看是否拥有这个变量。如果没有这个变量，就抛出错误，说明该变量没有被定义。

但是上面的设计还是有缺陷的，当 main 函数中拥有一个代码块如下：

```
int a;
int main () {
    int b,c;
    {int d};
    int e;
}
```

执行代码块的时候会创建 Variable 实例 v3，执行完成之后最新创建的实例是 v3，因此变量 e 会被放到 v3 中，而实际上它应该存在于 v2 中，因此需要一个栈式的结构，确保执行了代码块，或者是执行了函数调用，位于栈顶的实例总是正确的。实际上编译器对于函数调用也正

是使用这种方法。在 JavaScript 中没有栈这种数据结构，但是 JavaScript 的数字就拥有 push、pop 等 C 语言中栈所拥有的所有功能，因此可以使用数组替代。

创建一个保存变量实例的数组（栈）S，首先创建的全局变量实例先入栈 S，需要入栈的情况如下：

- 执行函数
- 进入代码块，For 循环中的代码块、If 中的代码块等只要被“{”和“}”包括的代码块

需要出栈的情况其实就是入栈时的情况执行完毕了，因此出栈对应着入栈的每一项，如下：

- 函数执行完毕
- 代码块执行完毕

理论上当一个 Variable 实例出栈后，且它不是任何实例的祖先，也就是其出栈后没有任何实例能够访问到它，那么这个实例就处于不可达的状态。可以将该实例销毁，释放其空间。

语句的执行

语句的执行采用的是对语法树进行前序遍历，遍历的顺序就是源程序执行的顺序。以下对一些具体的语句的执行进行说明，只是挑一些例子，并不包括所有的情况。

- 变量定义语句，假设文法如下：

LocalVarDecl -> Type id = id ;

当遍历到结点 LocalVarDecl 时，可以执行以下的操作了，由文法可知 Type 只能推到出 int、let 两种类型，因此可以通过寻找 Type 结点的子结点就能够得到具体的类型信息。同时“id”和“=”是终结符。直接可以获取“id”的 val，这里补充说明，“id”的 val 是如何被赋值的，进行语义分析前对语法树进行一次前序遍历，遍历的过程中对于遇到的叶子结点的顺利是源程序中 token 的顺序是一致的，因此可以将词法分析提供的词法值准确地赋值给终结符分 val。回到正题，文法中“=”左边的“id”是一个新定义的变量。而“=”右边的“id”的 val 属性可以是多种可能，联想到在词法分析中我们将数字的词法单元的第一个元素也分析成“id”，因此右边的“id”可以是字符串也可以是数字，还可能是一个变量，如果这个变量能够通过 Variable 实例找到，那么就是合法的，如果没有找到，就会抛出“变量没有定义”的错误。

- 赋值语句，赋值语句其实和变量定义语句十分相似。只是缺少变量的类型而已，因此进行赋值语句的执行是需要考虑到被赋值变量是否已经被定义，如果没有定义需要抛出错误。
- 函数调用语句，假设文法如下：

FunctionCallStmt -> id (AllParams) ;

首先对于 AllParams 的节点，可以使用遍历的方法获取到其所有参数，并存放在一个数组 array1 里。对于“id”，存在其在 Variable 实例中的值，确保这是一个已经定义的函数，或者是一个函数变量，如果其值是系统提供的函数，比如“console.log”类型，可以直接执行。而对于是源程序定义的函数，则肯定已经保存在栈底的全局变量 Variable 实例中，需要找到这个函数的值，并返回。本项目中，通过搜索函数名返回的结果是函数所在的结点的值。其结点的文法假设如下：

Function -> Type id (AllParams) Block

用同样的办法获取函数的 AllParams 结点下所有的变量，保存在数组 array2 中。不同的是，函数的参数列表中可以带有参数类型，因此有一点的差别。这样得到了 array1 和 array2，再将 array1 中每个元素的值赋值给 array2 中对应的参数。这样就可以对 Block 结点继续执行前序遍历了，也就是执行函数的内容。不要忘记上面提到的 Tree 数据结构中有个特殊的成员变量 returnVal，此时它将派上用场。执行完函数的内容后，如果有返回值，会将其赋值给

FunctionCallStmt 节点的 returnVal 变量。这样如果需要使用到函数的返回值，就可以获取。比如将函数的返回值打印输出。

- 布尔表达式，假设文法如下：

    BoolStmt -> BoolV CompareSign BoolV

布尔表达式相对来说简单，只要获取到布尔表达式两端的值，然后根据运算符进行即可。需要注意的一点是，BoolStmt 的 val 属性会被赋值为“true”或者“false”。

- For 循环语句，假设文法如下：

    ForStmt -> for (LocalVarDecl ; BoolStmt ; Statement) Block

For 循环语句只是看起来比较赋值，该文法右部的非终结符 LocalVarDecl、BoolStmt 已经分析过如何执行，而 Statement 一般来说都是赋值表达式，上面也讲述了执行的过程，因此对于 For 循环语句来说，关键点无非就是需要按照顺序正确的执行。本项目的实现方式是首先执行变量定义的语句，其次在 while 循环中执行 BoolStmt ，然后获取该结点的 val 值，之后执行 Block 里的内容，最后执行 Statement ，然后继续执行 BoolStmt ，知道 val 值为“false”才会结束 while 循环。因此 for 循环语句最重要的是语句的执行顺序。

还有很多的语句不一一解释说明，上面提到的语句已经覆盖到绝大多数的情况。

至此，语义分析的工作到此结束。代码解释器的工作就完成了。

第四章 项目的效果展示

第五章 总结与期望

本项目虽然是一个精简的代码解释器,仍然有几处值得提升的地方,比如可以根据文法,计算出每个非终结符的 First 集和 Follow 集，这样做的目的是可以使得在进行自顶向下的语法分析时，不会盲目的找到任何一个合适的文法就继续往下推导，如果需要进行回溯，会导致耗时更多，因此需要借助 First 集和 Follow 集对可以往下推导的文法进行筛选，虽然在本项目中可能没有起到很明显的作用,但是如果需要把本项目可支持的文法进行扩展时,在文法数量庞大的情况下会具有明显的效果,而自顶向下的语法分析方法随着文法数量的增加会出现性能急剧下降的情况。因为本项目文法相对简单，可以使用自动向下的分析方法。

文法相对简单,导致本项目另一个不足的地方是无法支持更多的语法。而本项目设计的初衷是建立一个初步的模型，使得 C 语言开发人员可以更加容易使用简单的 JavaScript，同时使得有这样需求的人可以使用本项目的模型，同时可以更快地进行扩展。

完成本项目，使得我对于编译原理的细节更加的了解，同时也更加明白“编程=数据结构+算法”这个道理。整个代码解释器完成的过程中，在不同的步骤，需要自己设计不同的数据结构，通过算法完成整个系统的设计。本项目算法都比较简单，自顶向下语法分析中用到回溯的方法。对于系统设计这种庞大的任务，如果数据结构能设计得很好，会让整个架构条理更加清晰，更加容易维护，更加容易扩展，甚至更加容易重构，可见其重要性。不单单是知识上的增长，单兵作战，一个人完成相对来说一个比较大的项目对于自己的编程能力也是有了不少的提升。同时在项目的设计和实现中也遇到了不少的困难，也使得自己发现问题和解决问题的能力得到极大提升，为以后的学习之路夯实了基础。